Implementation of Custom Precision Floating Point Arithmetic on FPGAs

Raj Gaurav Mishra and Amit Kumar Shrivastava
rajgmishra@hctl.org and amitshrivastava@hctl.org

Abstract

Floating point arithmetic is a common requirement in signal processing, image processing and real time data acquisition & processing algorithms. Implementation of such algorithms on FPGA requires an efficient implementation of floating point arithmetic core as an initial process. We have presented an empirical result of the implementation of custom-precision floating point numbers on an FPGA processor using the rules of IEEE standards defined for single and double precision floating point numbers. Floating point operations are difficult to implement on FPGAs because of their complexity in calculations and their hardware utilization for such calculations. In this paper, we have described and evaluated the performance of custom-precision, pipelined, floating point arithmetic core for the conversion to and from signed binary numbers. Then, we have assessed the practical implications of using these algorithms on the Xilinx Spartan 3E FPGA boards.
Keywords
Field Programmable Gate Array (FPGA), floating point arithmetic, custom-precision floating point, floating point conversion.

Introduction

With the advent of sensor technology, it is now possible to measure and monitor a large number of parameters and to carefully use them in a number of fields such as medical, defence, commercial etc. for various applications. Real-time implementation of sensor based application requires a system which can read, store and process the sensor data using micro-controllers or FPGAs as processors. Figure 1 shown below, represents a real-time data acquisition system based on a FPGA processor. Such a system comprises of a single or multiple sensors, signal conditioning unit (filters and amplifiers) and analog to digital converters. The output of the analog to digital converter is generally connected to the input of the processor (FPGA device in our case) for further signal acquisition and processing.

![Block diagram of a FPGA processor based real-time sensor data acquisition system.](image)

Figure 1: *Block diagram of a FPGA processor based real-time sensor data acquisition system.*

It is important for FPGA processor to store the real time sensor values to an external memory device for signal processing using custom algorithms. For example, analysing sound/speech in real time requires recording of sound signals using a microphone (sensor) using a high speed FPGA processor and then storing the resultant sensor values into a floating point format to maintain a specific accuracy and resolution. Floating point number system in comparison with binary number system have a better dynamic range and are better in
handling underflow and overflow situations during mathematical calculations (signal processing). In this way, when a sensor value is stored in floating point format provides a base for accurate signal processing.

In this paper, a pipelined implementation and hardware verification of custom precision floating point arithmetic on FPGA have been reported and discussed. Here, we assume that the reader is familiar with FPGA [1], its programming using Verilog HDL [2]-[3] and the single precision, double precision floating point standards [4] defined by IEEE. Comparatively, floating point number systems have a better dynamic range than a fixed point number system; also they are better in handling underflow and overflow situations during mathematical calculations however the speed and complexity issues arises when an implementation on FPGA processors comes into consideration. Research has been done to experiment various optimized implementations of IEEE single precision [7]-[10] and double precision [11]-[15] floating point arithmetic on FPGA. Algorithms for floating point implementation are complex in nature and with the number of bits used in single or double precision made them utilize a large area of the FPGA chip with a considerable processing time. Need of a custom precision floating point system arises when a real-time image or digital signal processing applications are to be implemented on an FPGA processor, where a requirement of high throughput in calculation and a balanced time-area-power implementation of the algorithm becomes an important requirement. In this way, an embedded designer can choose a suitable custom floating-point format depending upon the available FPGA space for the required embedded application.

Table 1 shows the basic comparison between the 17-bits custom precision, IEEE standards of single, double and quadruple precision floating point numbers.

This paper is organized as section 2 presents the custom precision floating point format in details. In section 3, we have described the algorithm and flowchart for the conversion of 12 bit signed binary number to 17 bits custom precision floating point number, the algorithm and flowchart for the conversion of 17 bits custom precision floating point number to a 12 bit signed binary number, along with their simulation results, synthesis summary and hardware verifications. Section 4 concludes this paper with the scope of future work which can be extended in various ways.
Table 1: Different floating point formats.

<table>
<thead>
<tr>
<th></th>
<th>Custom Precision</th>
<th>Single Precision</th>
<th>Double Precision</th>
<th>Quadruple Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Word Length</td>
<td>17 bits</td>
<td>32 bits</td>
<td>64 bits</td>
<td>128 bits</td>
</tr>
<tr>
<td>Mantissa</td>
<td>10 bits</td>
<td>23 bits</td>
<td>52 bits</td>
<td>112 bits</td>
</tr>
<tr>
<td>Exponent</td>
<td>6 bits</td>
<td>8 bits</td>
<td>11 bits</td>
<td>15 bits</td>
</tr>
<tr>
<td>Sign</td>
<td>1 bit</td>
<td>1 bit</td>
<td>1 bit</td>
<td>1 bit</td>
</tr>
<tr>
<td>Bias</td>
<td>$2^{6}-1=31$</td>
<td>$2^{8}-1=127$</td>
<td>$2^{11}-1=1023$</td>
<td>$2^{15}-1=16383$</td>
</tr>
<tr>
<td>Range</td>
<td>About $4.3 \times 10^{9}=(2^{32})$</td>
<td>About $3.5 \times 10^{38}=(2^{128})$</td>
<td>About $1.8 \times 10^{308}=(2^{1024})$</td>
<td>About $1.2 \times 10^{4932}=(2^{16384})$</td>
</tr>
</tbody>
</table>

Custom Precision Floating Point Format

Floating-point systems were developed to provide high resolution over a large dynamic range. Floating-point systems can often provide a solution when fixed-point systems, with their limited dynamic range, fail. Floating-point systems, however, bring a speed and complexity penalty. Most microprocessor floating-point systems comply with the published single- or double-precision IEEE floating-point standard; while in FPGA-based systems often employ custom formats.

A standard floating-point word consists of a sign-bit S, exponent E, and an unsigned (fractional) normalized mantissa M, arranged as shown in the figure 2.

![Figure 2: 17 bits Custom Precision Floating Point format.](image)

The minimum number custom precision floating point number (1,6,10) format can represent is:

Raj Gaurav Mishra and Amit Kumar Shrivastava
Minimum number representation in custom precision floating point format is \((0000000000000000)_{1,6,10}\) for a positive number and \((1000000000000000)_{1,6,10}\) for a negative number.

The maximum number custom precision floating point number \((1,6,10)\) format can represent is: \((±4,29,49,67,296.00)_{10}\) or \((±100000000000000000000000000000000)_{2}\).

Maximum number representation in custom precision floating point format is \((01111110000000000)_{1,6,10}\) for a positive number and \((11111110000000000)_{1,6,10}\) for a negative number.

\[
\begin{align*}
(0.0625)_{10} & \quad \text{Step 1: Take any decimal fraction number} \\
(0.0001)_{2} & \quad \text{Step 2: Convert the number in to a binary equivalent} \\
(1.0000 \times 2^{-4})_{2} & \quad \text{Step 3: Normalize according to the } 1.M \text{ format and get the value of } 2^{E} \\
(0000000000)_{2} & \quad \text{Step 4: Take the values of unsigned } M \text{ (mantissa) from the above } 1.M \text{ format. Normalize the value of mantissa } M \text{ as per the floating point format } (1,6,10). \\
\text{Bias } = 2^{e-1} - 1 = (31)_{10} & \quad \text{Step 5: Calculate the Bias by using the number of exponent bits allotted for the floating point format } (1,6,10). \\
E = -4 + \text{Bias } = (27)_{10} & \quad \text{Step 6: Calculate the value of Exponent } E, \text{ from the result generated from Step 3 and 5. Normalize the value of Exponent } E \text{ as per the floating point format } (1,6,10). \\
\end{align*}
\]

<table>
<thead>
<tr>
<th>S</th>
<th>Exponent (E)</th>
<th>Unsigned mantissa (M)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>011011</td>
<td>0000000000</td>
</tr>
</tbody>
</table>

**Figure 3:** Method of converting a fixed point decimal number in to a custom precision floating point number \((1,6,10)\) format [5]-[6].
Table 2: Some Examples values of 17-bit Custom Precision Floating-Point Format.

<table>
<thead>
<tr>
<th>S.No.</th>
<th>17-bit custom precision floating-point format</th>
<th>Equivalent decimal values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0 000000 0000000000</td>
<td>Represents a minimum number (+0)</td>
</tr>
<tr>
<td>2</td>
<td>1 000000 0000000000</td>
<td>Represents a minimum number (-0)</td>
</tr>
<tr>
<td>3</td>
<td>0 011111 0000000000</td>
<td>+1.0</td>
</tr>
<tr>
<td>4</td>
<td>1 011111 0000000000</td>
<td>-1.0</td>
</tr>
<tr>
<td>5</td>
<td>0 111111 0000000000</td>
<td>Represents a maximum number (+\infty)</td>
</tr>
<tr>
<td>6</td>
<td>1 111111 0000000000</td>
<td>Represents a maximum number (-\infty)</td>
</tr>
</tbody>
</table>

Floating Point Operations on FPGA

Signed Binary to Custom Precision Floating Point Conversion

Considering the system defined in the figure 1, an embedded designer can choose 8-bit, 12-bit or 16-bit analog to digital converters to adjust the required resolution of the sensor value for an application. Texas Instruments ADS7828 [16] or any other similar 12-bit ADC is more suitable for the algorithm developed and presented in the following section.

Algorithm 1 describes the step-wise approach of programming an FPGA for the conversion of a 12-bit signed binary number in to a 17-bit custom-precision floating point number.

The flow diagram of the algorithm 1 is shown in the figure 4. Table 3 shows the synthesis summary of hardware utilization and speed for the algorithm 1 on different FPGA processors. Figure 5 shows the simulation results for the algorithm 1 implemented using verilog HDL on Xilinx ISE Project Navigator Ver. 13.4 and ISE Simulator [17].

Custom Precision Floating Point to Signed Binary Conversion

Algorithm 2 defines the step-wise approach of programming an FPGA for the conversion of a 17-bit custom-precision floating point number in to a 12-bit signed binary number. The flow diagram of the algorithm 2 is shown in the
Algorithm 1 Converting a Signed Binary number (12 bits) in to a Custom-Precision (17 bits) Floating Point number (1,6,10) format.

**Require:** 12 bits signed binary number as input.

**Ensure:** 17 bit custom-precision floating point number (1,6,10) format as output.

1. Store the value of input to a temporary register R1 (size 12 bits).
2. Check for the sign-bit:
   3. if Sign-bit is equal to 1 (Input is a negative number): then Take 2’s complement of the values stored in R1 register (input value) and store the results to a temporary register R2 (size 12 bits).
   4. else if Sign-bit is equal to 0 (Input is a positive number): then Store the value of temporary register R1 to temporary register R2 without any modifications.
5. end if
6. Scan all the bit values of register R2 starting from (MSB \( -1 \)) towards LSB and search for first HIGH (1) bit value.
7. Count of the total bits towards right side (towards LSB) from the first HIGH (1) bit found, and store this count to a temporary register R3 (size 4 bits).
8. Store all the bits towards right side (towards LSB) from the first HIGH (1) bit found, to a temporary register R4 (size 10 bits).
9. Calculate the addition of the count stored in temporary register R3 with the value of fixed bias* and store the results to a temporary register R5 (size 6 bits). This forms the exponent value. Calculation of Fixed Bias* = \( 2^{E-1} - 1 = 2^{6-1} - 1 = 31_{10} \) or 011111₂. (E is the number of bits allocated for exponent in the floating point format).
10. Normalize the value of register R4 (bit shifting towards MSB to fit the values completely in 10 bits format). Store the resultant value to a temporary register R6. This forms the mantissa value.
11. Store the sign bit from the input value (as stored in register R1), value of register R5 (exponent) and value of register R6 (mantissa) in a 17 bits custom (1,6,10) format to a temporary register R7 (size 17 bits).
12. Connect the temporary register R7 to the output.
13. 17 Bit custom-precision floating point number (1,6,10) format.
Table 3: Synthesis Summary for 12 bit Signed Binary Number to 17 bit Custom Precision Floating Point Conversion.

<table>
<thead>
<tr>
<th>FPGA Processor</th>
<th>Speed Grade</th>
<th>Number of Slices Used</th>
<th>Number of Slice Flip Flops Used</th>
<th>Number of Slice Flip Flops Used</th>
<th>Number of 4 input LUTs Used</th>
<th>Number of bonded IOBs Used</th>
<th>Maximum Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spartan 3E</td>
<td>-5</td>
<td>71 out of 4656</td>
<td>70 out of 9312</td>
<td>134 out of 9312</td>
<td>30 out of 232</td>
<td>174.304 MHz</td>
<td></td>
</tr>
<tr>
<td>XC3S1200E</td>
<td>-5</td>
<td>71 out of 8672</td>
<td>70 out of 17344</td>
<td>134 out of 17344</td>
<td>30 out of 250</td>
<td>174.304 MHz</td>
<td></td>
</tr>
<tr>
<td>Spartan 6</td>
<td>-3</td>
<td>77 out of 30064</td>
<td>138 out of 15032</td>
<td>162 out of 486</td>
<td>30 out of 226</td>
<td>212.770 MHz</td>
<td></td>
</tr>
<tr>
<td>XC6SLX25</td>
<td></td>
<td>30064</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtex 4</td>
<td>-12</td>
<td>83 out of 42176</td>
<td>70 out of 84352</td>
<td>157 out of 84352</td>
<td>30 out of 576</td>
<td>338.324 MHz</td>
<td></td>
</tr>
<tr>
<td>XC4VFX100</td>
<td></td>
<td>42176</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtex 5</td>
<td>-3</td>
<td>70 out of 64000</td>
<td>94 out of 64000</td>
<td>113 out of 339</td>
<td>30 out of 680</td>
<td>394.120 MHz</td>
<td></td>
</tr>
<tr>
<td>XC5VFX100T</td>
<td></td>
<td>64000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtex 6</td>
<td>-2</td>
<td>69 out of 160000</td>
<td>120 out of 80000</td>
<td>136 out of 408</td>
<td>30 out of 240</td>
<td>433.529 MHz</td>
<td></td>
</tr>
<tr>
<td>XC6VCX130T</td>
<td></td>
<td>160000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4 shows the synthesis summary of hardware utilization and speed for the algorithm 2 on different FPGA processors. Figure 6 shows the simulation results for the algorithm 2 implemented using verilog HDL on Xilinx ISE Project Navigator Ver. 13.4 and ISE Simulator [17].

Algorithms 1 and 2 have been tested and verified on Digilent NEXYS 2 FPGA board [18] containing Spartan 3E [19] XC3S1200E FPGA processor as shown in figure 7. To verify the correctness of the algorithm different inputs were given through different combinations of on-board switches and output was received through the LEDs connected to the I/O pins of the FPGA processor.
Algorithm 2 Converting a Custom-Precision (17 bits) Floating Point number (1,6,10) format in to a Signed Binary number (12 bits).

**Require:** 17 bit custom-precision floating point number (1,6,10) format as input.

**Ensure:** 12 bits signed binary number as output.

1: Store the MSB value of input to a temporary register \( R1 \) (size 1 bit). This is for the purpose of defining sign bit.
2: Store the 10 bits from the LSB towards MSB to a temporary register \( R2 \) (size 11 bits). This is to be used as mantissa for further calculations.
3: Assign the MSB bit of temporary register \( R2 \) as HIGH (1) to incorporate the hidden 1 bit.
4: Store the remaining 6 bits from the input to a temporary register \( R3 \) (size 6 bits). This is to be used as exponent for further calculations.
5: Calculation of exponent value (in order to normalize the mantissa): Values stored in register \( R3 - \text{Bias}^* = \text{Value of exponent by which mantissa is to be normalized.} \) Store this value to a temporary register \( R4 \) (size 8 bits). Calculation of Fixed Bias* = \( 2^E - 1 = 2^{10} - 1 = 31_{10} \) or 011111 \( _2 \). (\( E \) is the number of bits allocated for exponent in the floating point format).
6: Normalization of mantissa value to fit in 11 bits:
7: if the value stored in register \( R3 \) (exponent value) is equals to 0 (zero). then the value of mantissa will become 0 (zero).
8: end if
9: if the value stored in register \( R4 \) (exponent count) is equals to 0. then the value of mantissa is to be bit-shifted 10 times towards left (MSB) for normalization.
10: end if
11: if the value stored in register \( R4 \) (exponent value) is equals to 1. then the value of mantissa is to be bit-shifted 9 times towards left (MSB) for normalization.
12: end if
13: if the value stored in register \( R4 \) (exponent value) is equals to 2. then the value of mantissa is to be bit-shifted 8 times towards left (MSB) for normalization.
14: end if
15: if the value stored in register \( R4 \) (exponent count) is equals to 3. then the value of mantissa is to be bit-shifted 7 times towards left (MSB) for normalization.
16: end if
17: if the value stored in register \( R4 \) (exponent value) is equals to 4. then the value of mantissa is to be bit-shifted 6 times towards left (MSB) for normalization.
18: end if

Raj Gaurav Mishra and Amit Kumar Shrivastava

Implementation of Custom Precision Floating Point Arithmetic on FPGAs.
if the value stored in register R4 (exponent value) is equals to 5. then
the value of mantissa is to be bit-shifted 5 times towards left (MSB) for
normalization.
end if

if the value stored in register R4 (exponent value) is equals to 6. then
the value of mantissa is to be bit-shifted 4 times towards left (MSB) for
normalization.
end if

if the value stored in register R4 (exponent value) is equals to 7. then
the value of mantissa is to be bit-shifted 3 times towards left (MSB) for
normalization.
end if

if the value stored in register R4 (exponent value) is equals to 8. then
the value of mantissa is to be bit-shifted 2 times towards left (MSB) for
normalization.
end if

if the value stored in register R4 (exponent value) is equals to 9. then
the value of mantissa is to be bit-shifted 1 time towards left (MSB) for
normalization.
end if

Store the resultant value of mantissa after suitable bit-shifting (normalization) to a temporary register R5 (size 11 bits).

Check for the sign-bit stored in the register R1:

if Sign-bit is equal to 1 (Input is a negative number): then Take 2’s complement of the values stored in R5 register (normalized value of mantissa) and store the results to a temporary register R6 (size 12 bits).
else if Sign-bit is equal to 0 (Input is a positive number): then Store the value of temporary register R5 to temporary register R6 without any modifications.
end if

Connect the temporary register R6 to the output.

12 bits signed binary number.
Table 4: Synthesis Summary for 17 bit Custom Precision Floating Point to 12 bit Signed Binary Number Conversion.

<table>
<thead>
<tr>
<th>FPGA Processor</th>
<th>Speed</th>
<th>Number of Slices Used</th>
<th>Number of Slice Flip Flops Used</th>
<th>Number of 4 input LUTs Used</th>
<th>Number of bonded IOBs Used</th>
<th>Maximum Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spartan 3E XC3S500E</td>
<td>-5</td>
<td>56 out of 4656</td>
<td>47 out of 9312</td>
<td>100 out of 9312</td>
<td>30 out of 232</td>
<td>199.222 MHz</td>
</tr>
<tr>
<td>Spartan 3E XC3S1200E</td>
<td>-5</td>
<td>56 out of 4656</td>
<td>47 out of 9312</td>
<td>100 out of 9312</td>
<td>30 out of 232</td>
<td>199.222 MHz</td>
</tr>
<tr>
<td>Spartan 6 XC6SLX25</td>
<td>-3</td>
<td>48 out of 30064</td>
<td>72 out of 15032</td>
<td>90 out of 486</td>
<td>30 out of 226</td>
<td>248.738 MHz</td>
</tr>
<tr>
<td>Virtex 4 XC4VFX100</td>
<td>-12</td>
<td>55 out of 42176</td>
<td>47 out of 84352</td>
<td>99 out of 84352</td>
<td>30 out of 576</td>
<td>340.833 MHz</td>
</tr>
<tr>
<td>Virtex 5 XC5VFX100T</td>
<td>-3</td>
<td>47 out of 64000</td>
<td>61 out of 64000</td>
<td>80 out of 339</td>
<td>30 out of 680</td>
<td>440.567 MHz</td>
</tr>
<tr>
<td>Virtex 6 XC6VCX130T</td>
<td>-2</td>
<td>47 out of 160000</td>
<td>72 out of 80000</td>
<td>90 out of 408</td>
<td>30 out of 240</td>
<td>461.563 MHz</td>
</tr>
</tbody>
</table>

Conclusion and Future work

We have successfully implemented and tested the functionality of custom precision floating point numbers on FPGAs. The main objective of this research is to develop and implement a real-time sensor data acquisition system based on FPGA. In order to achieve it, the following activities are planned be carried out in future: Implementation of Multiplication, Addition/Subtraction and Division algorithms on custom precision numbers on FPGAs, Implementation of I2C protocol to read serial ADC data on FPGA, Implementation of SD card and display module on FPGA to store and display real-time sensor data. These algorithms would be helpful in handling physical connections, storage and display of incoming sensor data and implementation of some basic digital signal processing techniques.
References


This article is an open-access article distributed under the terms and conditions of the Creative Commons Attribution 3.0 Unported License (http://creativecommons.org/licenses/by/3.0/).

© 2013 by the Authors. Licensed & Sponsored by HCTL Open, India.
Figure 4: Flow diagram for conversion of a 12 bit Signed Binary Number in to a 17 bit Custom Precision Floating Point Number - Pipelined approach
Figure 5: Simulation Results for 12 bit Signed Binary Number to 17 bit Custom Precision Floating Point Conversion

Figure 6: Simulation Results for 17 bit Custom Precision Floating Point to 12 bit Signed Binary Number Conversion

Figure 7: Digilent NEXYS 2 FPGA board hardware setup for algorithm testing and verification

Raj Gaurav Mishra and Amit Kumar Shrivastava
Implementation of Custom Precision Floating Point Arithmetic on FPGAs.
Figure 8: Flow diagram for conversion of a 17 bit Custom Precision Floating Point Number in to a 12 bit Signed Binary Number - Pipelined approach